Random Test Run Length and Effectiveness

James H. Andrews[*], Alex Groce[†], Melissa Weston[*], and Ru-Gang Xu[‡]
[*] Dept. of Computer Science, University of Western Ontario (andrews@csd.uwo.ca, mweston2@csd.uwo.ca)
[†]Laboratory for Reliable Software, Jet Propulsion Laboratory, California Institute of Technology (Alex.Groce@jpl.nasa.gov)
[‡]Dept. of Computer Science, University of California, Los Angeles (rxu@cs.ucla.edu)

A portion of the research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

***Abstract*—A poorly understood but important factor in random testing is the selection of a maximum length for test runs. Given a limited time for testing, it is seldom clear whether executing a small number of long runs or a large number of short runs maximizes utility. It is generally expected that longer runs are more likely to expose failures — which is certainly true with respect to runs shorter than the shortest failing trace. However, longer runs produce longer failing traces, requiring more effort from humans in debugging or more resources for automated minimization. In testing with feedback, increasing ranges for parameters may also cause the probability of failure to decrease in longer runs. We show that the choice of test length dramatically impacts the effectiveness of random testing, and that the patterns observed in simple models and predicted by analysis are useful in understanding effects observed in a large scale case study of a JPL flight software system.**

I. Introduction

Random testing, an approach in which test inputs are generated at random (with a probability distribution that may change as testing proceeds, and usually with the possibility that inputs may be generated more than once), has recently been shown to be an effective and easy-to-use automatic test generation technique for a wide variety of software. This software can be divided into two categories [1]: *batch* or *interactive*. Batch programs take a single input, such as a string, and return an output. Interactive programs take a *sequence* of inputs (typically including a choice of *operation*, usually a function or method call) that may change the state of the program, affecting output for future inputs. Many safety critical programs, such as operating systems, network applications, and control systems fall into this category. As in all testing, the goal of random testing is to produce test *failures*: test cases in which a program *fault* (a particular bug, repaired by a particular fix) induces error in program state that propagates to observable output.

Recent work on random testing has focused on strategies for testing interactive programs, including file systems [2], data structures [3], [4], [5], and device drivers. For such programs, a *random test suite* is a set of *test runs*. Each test run is described by a sequence of operations performed starting from a fixed initial program state. A test budget (the time available for testing, approximated by limiting the number of operations) is typically divided into more than one test run, as failures can result from one-time decisions made at the beginning of a run. Intuitively, testers expect that using the entire test budget for a single run and re-initializing the program state after every operation (performing many runs of length one) are both unwise strategies, but little more is known about how to divide a budget. Testers may assume that longer runs (up to some point short of a single run) will be more likely to expose faults, motivated in part by the fact that for every program there is some (unknown) shortest failing trace and that no shorter run can fail, but often have little empirical support for this suspicion. Even random testing researchers often choose a length based on little more than an educated guess and do not experiment with this *ad hoc* choice [2].

It is also assumed that longer runs will produce longer failing traces, which are more difficult to analyze and more expensive as regression tests. In particular, with random testing, long runs will contain a large number of irrelevant operations, hindering debugging. Automated test case minimization via delta-debugging [6] can reduce long traces to a more manageable length, and is essential for making random testing useful [7]. Delta-debugging performs a kind of binary search, potentially quadratic in the test length, to find a shorter 1-minimal test case (a failing test case that succeeds if any operation is removed). While quadratic behavior is seldom observed, the cost of delta-debugging does indeed increase with test length, and (because it finds a 1-minimal rather than globally minimal trace) delta-debugging will tend to produce longer minimized traces from longer runs.

In this paper, we demonstrate that the length of test runs does indeed significantly impact the number of failures discovered as well as the length of failing traces found, and may be a major factor in random test effectiveness.

Are Longer Runs Better at Finding Failures? In a limited sense, longer runs *are* always better, under two assumptions:
Assumption 1: Checks for failure are performed after every step of a sequence, rather than only at the end of the sequence.
Assumption 2: The probability that a generated test run of length $k+1$ will have a certain prefix of length k is the same as the probability of generating that run of length k.

If both assumptions hold, then a test run of length $k+1$ will necessarily have a probability of failure equal to or greater than that of a test case of length k. That is, if our random test consists of one test run and we aim to maximize the probability of failure, it should be as long as possible, if the maximum length does not affect the selection of test operations — even if the probability of failure *decreases* with each step.

In reality, as noted above, software is tested in limited time and with more than one run. A more realistic model is to consider how a test budget should be partitioned into runs. Given a fixed budget of B operations, choosing a length k determines how many runs will be executed — ranging from one run of length B to B runs of length one. While the actual cost in machine time of test operations may vary, controlling testing time by fixing a budget of operations is reasonable: the choice to terminate a test usually cannot be made in mid-operation, and if operations are equally probable the average cost of k-length tests is often predictable. In some experiments and analysis, we assume that even if a test terminates early after detecting a failure, k operations are still counted against the budget. This is not unreasonable, as the purpose of the budget is to limit resources, and the cost of minimizing the failing test case is likely to be *greater* than the cost of the remaining operations. In this model, increasing run length can decrease the effectiveness of a test effort, as the expected number of failures depends on the number of runs executed. Let $P(k)$ be the probability of finding a failure at length k. The total number of failing traces found with a test budget of B operations with length k test runs is $N(k) = \left\lfloor \frac{B}{k} \right\rfloor \cdot P(k)$. The expected number of traces found increases when we increase length from k_1 to k_2 iff $\left\lfloor \frac{B}{k_2} \right\rfloor \cdot P(k_2) > \left\lfloor \frac{B}{k_1} \right\rfloor \cdot P(k_1)$. That is, if we double the length of runs, we *lower* the expected number of failing traces, unless the probability of failure doubles.

Related Work. Although many random testing strategies have been proposed, factors that influence the effectiveness of all such test strategies have not been deeply explored. Only recently has the effect of the seed and timeout been investigated thoroughly [8]. Whittaker and Thomason propose using a Markov chain model in stochastic testing, but do not address factors for selecting run length [9]. Doong and Frankl [10] also noted that different numbers of operations in random testing resulted in different failure rates.

Contributions. This paper presents the first large empirical study of how test run length affects failure detection and trace quality. In our simple examples and larger case studies, we show that there often exists an optimal length of input sequences for failure detection, and that longer runs lead to longer failing traces and more time spent minimizing tests. We begin by examining two small models, providing an intuitive understanding of the effects of run length. We then show how these effects appear during the large-scale testing of an embedded file system at JPL and in unit testing of Java data structures. We focus on how run length affects failure detection — how the number of failures discovered for a given testing budget varies with test run length. Of course, discovering many traces exposing the same fault is not the goal of testing (though it can be useful in evaluating fixes). However, in cases where there is only one fault in a system (or a small number of faults of roughly equal probability), *failure detection for large budgets serves to approximate the chance of finding any failure at all (and thus any fault) with a smaller budget*, and is a simpler statistic to compute and understand than expected-probability-of-finding-a-fault (which is easy to derive from failure detection in our examples).

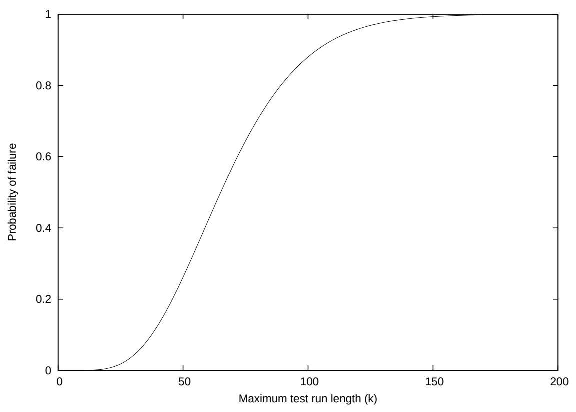

Fig. 1. Probability of failure, Buffer overflow

The particular effects of run length vary with program (and version of that program): we here present a *study of those effects* rather than *a method for selecting optimal run length*. We do not, therefore, study the large range of "mixed" strategies for dividing a budget in which k is not constant but varies over time. We do note that an "iterative deepening" method of starting with small k and increasing it until failure detection rates decrease might be useful.

Threats to Validity. There is some possibility that our 5 programs (with 9 versions for one) might be unrepresentative. In particular, effects of run length might not be similar for programs with very low failure incidence. Results for low failure versions of the file system do not contradict our findings, but producing statistically significant results for such programs appears to be prohibitively expensive.

II. Examples

For interactive programs, a sequence of operations may lead to failure. For many faults, there exists a finite set of minimized failure traces E — minimal sequences that expose the fault. For example, one bug in the JPL file system is described by minimized traces of only 2 operations, one `mkdir` and one `open`. If the random tester has a finite set of operations M to choose from at each step of the test run, then the set of all possible test runs or traces of length k or less, T_k, is also finite. From the set of minimized failing traces, a finite set of failing traces of length less than or equal to k, F_k, can be derived by including all operations that do not contribute to (or prevent) failure for each minimized failure trace. Given a random trace of length k, there is a $P(k) = |F_k|/|T_k|$ probability of failing.

For this category of programs and faults, there exists some finite optimal run length. To simplify calculations, we assume that at each step the tester chooses an operation from M uniformly. For each minimized failing trace ρ, we can calculate the number of traces of length k that contain the failing trace as $B(k, |\rho|) = \binom{k}{|\rho|}(|M| - b)^{k-|\rho|}$, where b are the unique operations in ρ. The probability of finding failure for length k is thus $P(k) = \frac{\sum_{\rho \in E} B(k,|\rho|)}{|M|^k}$.

An example of the probability function P is shown in Figure 1. All interactive programs with this class of faults

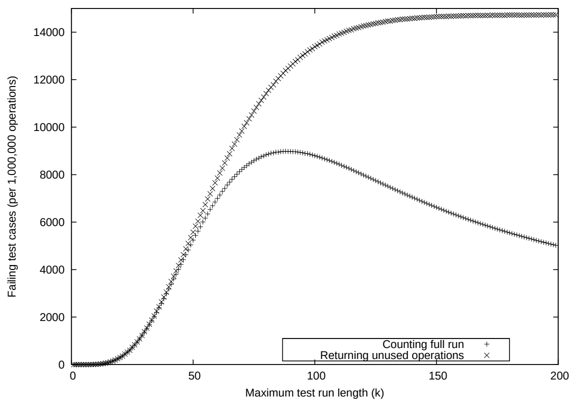

Fig. 2. Failure detection with and without the testing budget counting the full run, Buffer overflow

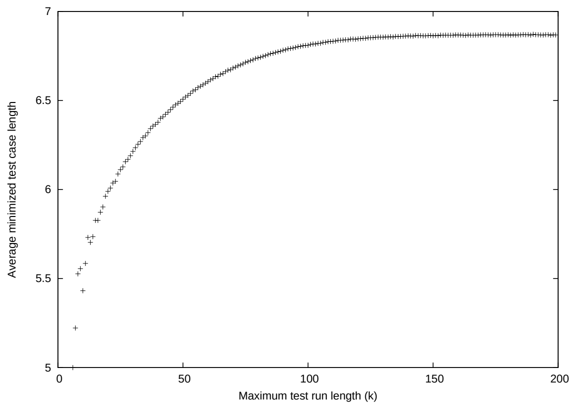

Fig. 4. Avg. minimized failing trace length, Buffer overflow

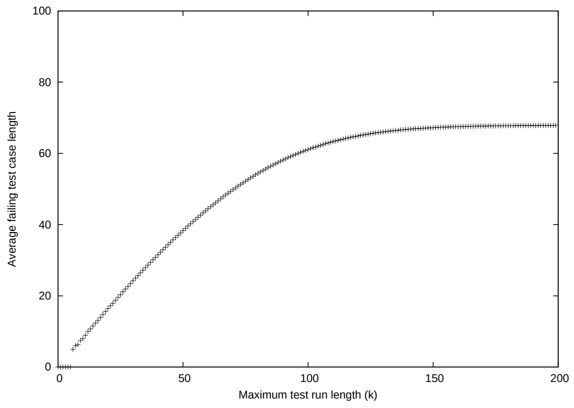

Fig. 3. Avg. failing trace length, Buffer overflow

will produce a similar binomial cumulative distribution curve. From this distribution, the existence of some optimal test run length can be predicted, by calculating $P(k)/k$. We precisely calculate the probability for a simple example of buffer overflow to demonstrate that this calculation accurately predicts the optimal test run length. Case studies, although we cannot precisely calculate P, show similar behavior, hinting that real experiments exhibit the properties described by our simple examples.

A. Buffer Overflow

This example shows how we can predict how the length of test runs affects the probability of failure where each operation is uniformly selected. This example is representative of more complex failures found in our case studies, where a certain sequence of operations must be performed before failure. Suppose there are 10 buffers, one of which is incorrectly allocated and overflows as soon as a 10th item is added to it. We can “test” this system with a very simple driver:

```
for (i = 0; i < k; i++) {
  j = rand() % 10;
  amount = rand() % 2 + 1;
  write (buffers[j], amount);
  assert (buffers[BADBUFF] < 10);
}
```

At each step, one or two items are written to a random buffer. The simulation fails if the bad buffer has 10 or more items. We stop a run after k steps. A *trace* is a sequence of writes (b, n) where b is the buffer written and $n \in \{1, 2\}$ is the number of items written. A *failing trace* is a trace that ends with the assertion failure. Failing traces contain writes to the bad buffer and may contain writes to other buffers. If the writes to the other buffers are removed, the trace is a *minimized failing trace*. Ideally, we want to find a test length k such that we find the most failing traces, and such that all failing traces or all minimized failing traces have the shortest possible length.

Failure Detection. To calculate the probability of failing given a test case of length k, we calculate $P(k) = F_k/T_k$. We now define $B(k, s) = \binom{k}{s}(18)^{k-s}$ as the number of traces that contain a sequence of s writes to a bad buffer of length k. There are 10 classes of traces that do not lead to failure: traces that do not have writes to the bad buffer ($s = 0$) and traces that insert $s = 1 \ldots 9$ items in the bad buffer. Figure 1 shows the plot of $P(k)$ for k between 1 and 200.

To find the k that maximizes failure detection, we want to increase k_1 to k_2 if and only if $P(k_2)/k_2 > P(k_1)/k_1$. $P(k)/k$ is maximized when $k = 92$. Therefore, the optimal run length should be 92. Figure 2 shows failures found per 1,000,000 operations by using random tests with a budget of 10,000,000 operations with run lengths from 1 to 200. As predicted, the graph peaks when k is 92. In one experiment, if there is an assertion failure in the beginning of the test run, we continue the test and count all operations performed. In a second experiment we terminate the test run at the assertion failure and do not count the subsequent operations toward the testing budget. In this case, failures found per operation stabilize to some constant as we increase the input length, confirming the notion that longer runs result in better failure detection.

Failing Trace Length. There are 144 minimized failing traces: 89 that result in 10 items being in the buffer and 55 where the final insertion adds two items to the buffer, for a total of 11. The smallest minimized failing trace is $(bad, 2) : (bad, 2) : (bad, 2) : (bad, 2) : (bad, 2)$; the longest

$$\begin{array}{l} A \rightarrow B : \{N_A, A\}_{K_{PB}} \\ B \rightarrow A : \{N_A, N_B\}_{K_{PA}} \\ A \rightarrow B : \{N_B\}_{K_{PB}} \end{array} \qquad \begin{array}{l} A \rightarrow I : \{N_A, A\}_{K_{PI}} \\ I \rightarrow B : \{N_A, A\}_{K_{PB}} \\ B \rightarrow I : \{N_A, N_B\}_{K_{PA}} \\ I \rightarrow A : \{N_A, N_B\}_{K_{PA}} \\ A \rightarrow I : \{N_B\}_{K_{PI}} \\ I \rightarrow B : \{N_B\}_{K_{PB}} \end{array}$$

Fig. 5. **Needham-Schroeder (NSPK):** (a) Protocol (b) Man-in-the-middle attack

is a trace of 9 writes of 1 item to the bad buffer followed by $(bad, 1)$ or a $(bad, 2)$. Figure 3 shows the average failing trace length for $k = 1...200$ with a budget of 10^7 operations. Figure 4 shows the average minimized failing trace length. In both graphs, $k = 92$, the optimum choice for failure detection, has an average (minimized) failing trace length close to the maximum length, hinting that there is a trade-off between the ability to detect faults and the quality of traces.

B. Needham-Schroeder

This example shows how there exists an optimal test run length when operation selection is not uniform — e.g., when the probability that an operation leading to error is selected decreases as the test run increases. The Needham-Schroeder Public-Key Protocol (NSPK) [11] provides authentication between two parties A and B. A sends B a nonce and A's identity encrypted by B's public key. B replies with A's nonce and a newly generated nonce encrypted by A's public key. A replies with B's nonce encrypted with B's public key. At the end of the protocol, B is supposed to know that A is indeed talking with B and vice versa. Unfortunately, there is a known man-in-the-middle attack [12]. An impostor I can initiate the protocol and induce B to believe that the protocol is executing with A rather than I. I operates by forwarding messages to A, since I never needs to encode its identity with A's key. The protocol and the attack are shown in Figure 5.

Random testing can find this man-in-the-middle by modeling two legitimate parties, A and B, and a random adversary R on a shared network. A and B will always follow the protocol: each may randomly choose some party and begin authentication. If A chooses B, authentication will occur as in Figure 5. If either A or B chooses R, A or B will ignore messages that do not follow the protocol. A or B will reset after receiving n unexpected messages. R randomly generates message from communications overheard and randomly selects a receiver. R does not know the protocol, but can decrypt messages encrypted by its public key and assemble new messages. This model represents a realistic approach to randomly testing a protocol for a wide variety of attacks.

Failure Detection. Figure 6 shows how the length of each run affected authentication failure detection with a test budget of 1,000 operations. The y-axis shows failing traces found per 100 operations. We find that the most effective length is 50 operations. Increasing past 50 decreases the effectiveness, because R has too many recorded messages to choose from. Again, we show results when operations after failure are both counted against the total and returned to the test budget. This is a simple example where feedback [3], [2] influences failure detection: as a run accumulates history, the range of the random choices increases, and at some point the probability of failing (by matching a nonce) begins to diminish. In this case, the intuition that longer test runs find more failing traces is incorrect.

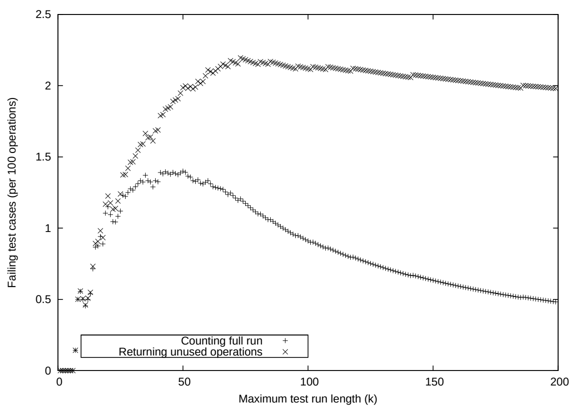

Fig. 6. Failure detection with and without the testing budget counting the full run, NSPK

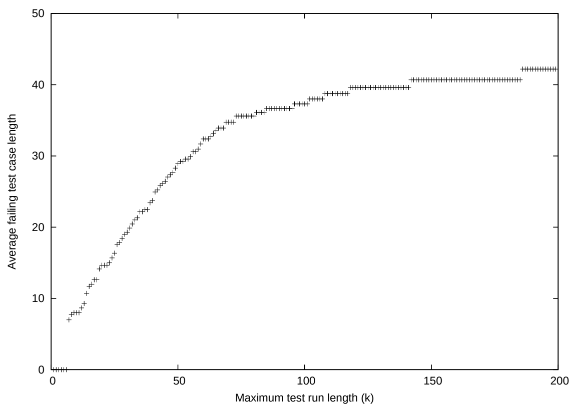

Fig. 7. Avg. failing trace length, NSPK

Failing Trace Length. In this case, the final result of minimization is pre-determined: there is one unique failing trace (abstracting nonces) leading to the exploit (Figure 5(b)). Figure 7 shows how k affected the failing trace length. The optimum test run length for failure detection produces failing traces with an average of 29 operations, significantly more than the minimum of 6 operations. Very small k produce near-optimal failing traces but are less likely to produce failures.

III. Case Study: Flash File System

The results in this section were generated using a random test framework for file systems used in space missions [2] (which we are now applying to file systems for the Mars Science Laboratory [13]). We selected versions of the file system and framework ranging from the earliest working versions to stable versions almost identical to the current release (identified by date from 01-19-2006 to 09-06-2006). Failure density ranges from very low (two failing test cases

	01-19	02-03	02-17	03-03	03-17	04-03	04-27	04-28	09-06
Failure detection	8	9	10	11	11	11	12	S	S
Failure rates	13	14	14	15	15	15	14	S	S
Avg. failing trace length	16	17	17	17	17	17	17	S	S
Minimization cost	18	C	C	18	18	18	C	18	18
% minimization cost	19	C	C	19	19	19	C	19	19
Avg. minimized trace length	20	C	C	20	20	20	C	L	L
Shortest minimized trace length	L	C	C	L	L	L	C	21	21

Reasons for omission: L = lack of interesting features, C = computation-time limits, S = too few points for significance

TABLE I
FILE-SYSTEM GRAPH OVERVIEW (NUMBERS ARE REFERENCES TO FIGURES)

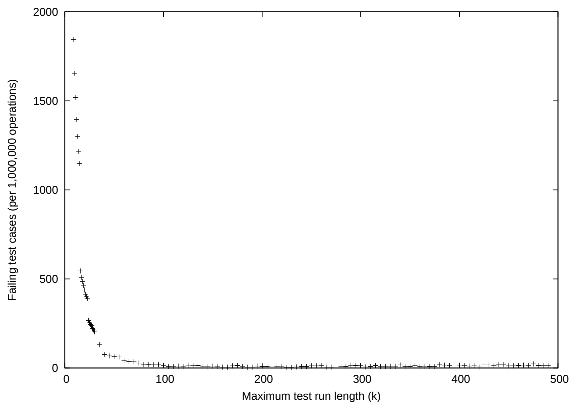

Fig. 8. Failure detection, 01-19

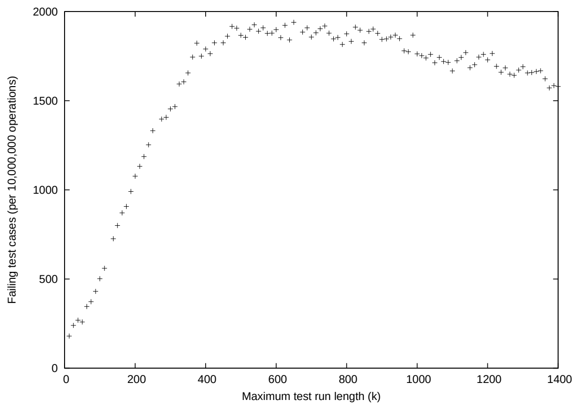

Fig. 10. Failure detection, 02-17

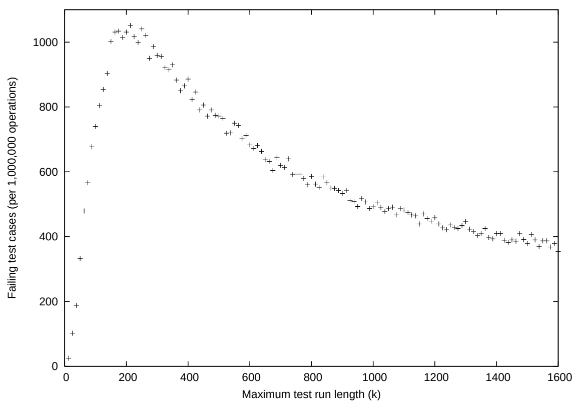

Fig. 9. Failure detection, 02-03

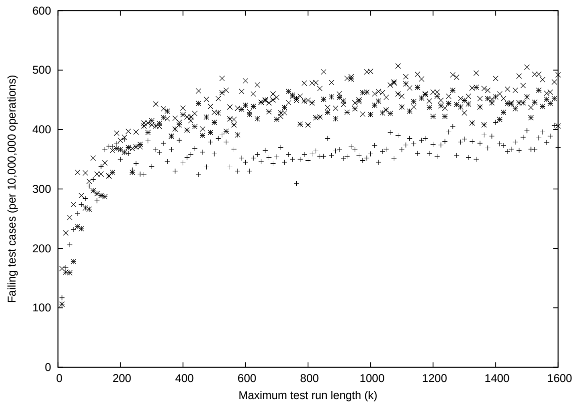

Fig. 11. Failure detection, 03-03, 03-17, 04-03

in two days of testing) to very high (thousands of failing test cases per hour).

Our results are based on further execution of approximately 25 billion test operations (over 60 machine days). For each version, depending on the density observed during logged tests, we ran with a testing budget of 1, 10, or 100 million operations and a run length k ranging in 128 even steps from 13 to 1,600 (in most cases). In some cases, we only examined test lengths up to 1,400, as the file system releases for these versions were compiled with resource limitations that caused false warnings to dominate the test results with longer tests (the effect does not appear in shorter tests, but false warnings become difficult to filter out with longer tests). For other versions, this problem did not appear until a length of around 1,600. For 01-19, we observed convergence to very few failures at low test lengths, and increased the number of sample points for lower values to better show this behavior. For each k, we recorded (1) the number of failing test cases produced. For some versions we also computed (2) the average failing test case length, (3) the lengths of minimized failing traces produced from the failing test cases, and (4) the number of operations spent minimizing test cases. An operation, for testing or minimization, took an average ranging from about 0.0002 to 0.0004 seconds to execute, depending on the version of the file system. Failures represent one fault or two faults of approximately equal probability, to the best

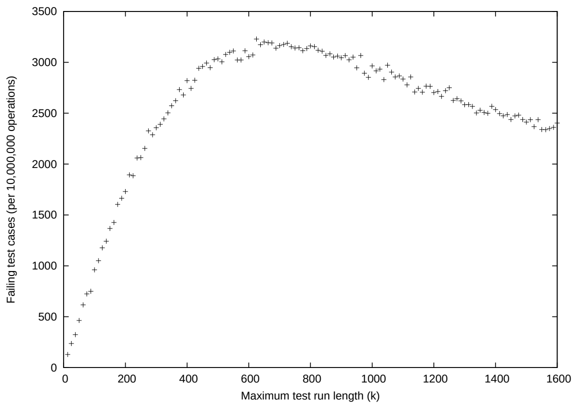

Fig. 12. Failure detection, 04-17

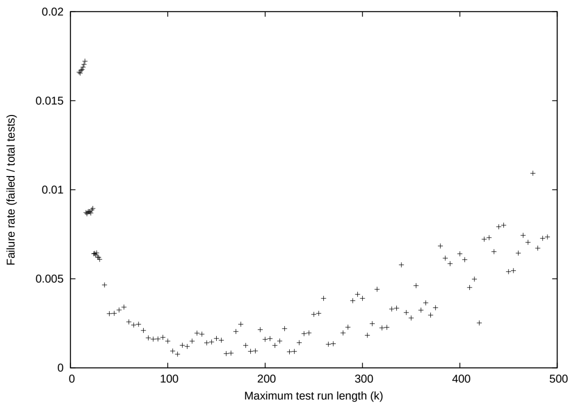

Fig. 13. Failure rates, 01-19

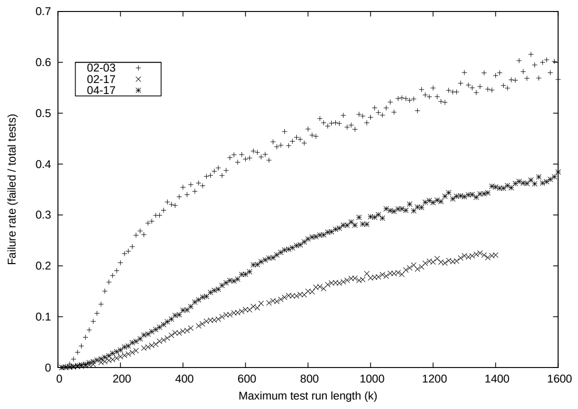

Fig. 14. Failure rates, 02-03, 02-17, 04-17

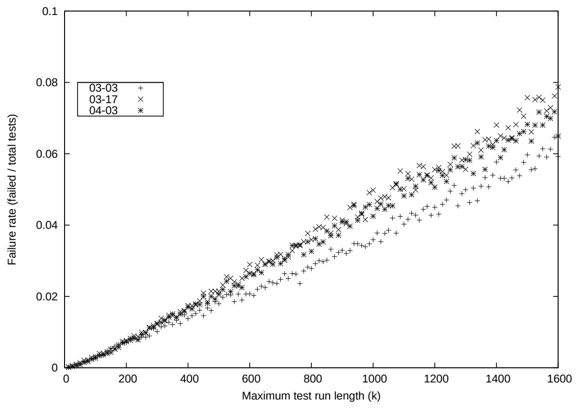

Fig. 15. Failure rates, 03-03, 03-17, 04-03

of our knowledge, in all but one case. Thus failure detection approximates probability of finding all faults, as desired. Table I gives an overview of locations for results, indicating those omitted due to a lack of interesting features (L), computation-time limits (C), or because there are too few data points for significance (S).

A. Failure Detection

Figures 8-12 show the number of failing test cases for each choice of the test run length k. As noted in earlier work, the number of unique faults (identified by bug fixes) decreased with time [2]. These figures show that failure detection for test periods with one or two faults also generally decreased as the software grew more stable: there were fewer bugs and the bugs were (usually) less likely to occur in any test run. The effect was most marked at the beginning of the test period (where for small k the detection rate was nearly 2,000 failures per 1,000,000 random operations) and at the end of testing (0.08 failures per 1,000,000 operations). From 03-03 to 04-03, failure detection remained fairly constant, before peaking again then stabilizing very low. One observation is that optimizing failure detection was usually unimportant during early testing, as it was easy to find faults. For later versions, we were fortunate that our (*ad hoc*) selection of $k = 1,000$ was close to the optimal.

The effect of k on failure detection is unclear for the final versions of the software, 04-28 and 09-06, where the probability of failure is too low to show any meaningful trends. With a test budget of 100 million operations, testing never produced more than 8 failing test cases for any choice of k. We speculate that trends might be evident for these versions if we increased the budget to 10 billion or more operations (but estimate that producing these results would take at least 2,000 machine days, a daunting prospect even given the embarrassingly parallel nature of random testing).

As Figures 14 and 15 show, the *rate* of failure for tests was still increasing at the point at which false warnings forced us to end our experiments. This appeared to be the case for 04-28 and 09-06, though in these cases the infrequency of failures made it difficult to be certain. In three cases, the increase in failure rate was sufficient to make the failure detection appear roughly constant, while in other cases the rate had decreased enough to produce a decrease in failure detection. Figure 13 shows a fundamentally different pattern (and demonstrates that the file system test framework can violate our **Assumption 2**). **Behavior for Larger k.** False positives cause the “failure rate” to approach 1.0 quickly after a certain test length is reached. At this point experiments show only that increasing k means performing fewer tests.

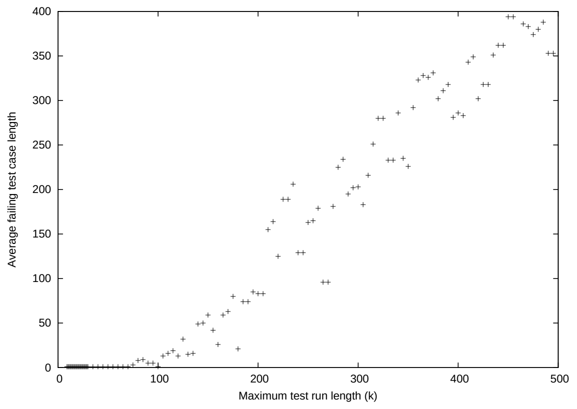

Fig. 16. Avg. failing trace length, 01-19

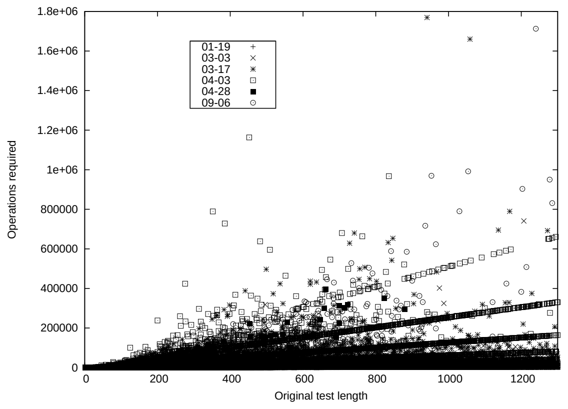

Fig. 18. Minimization cost, 01-19, 03-03, 03-17, 04-03, 04-28, 09-06

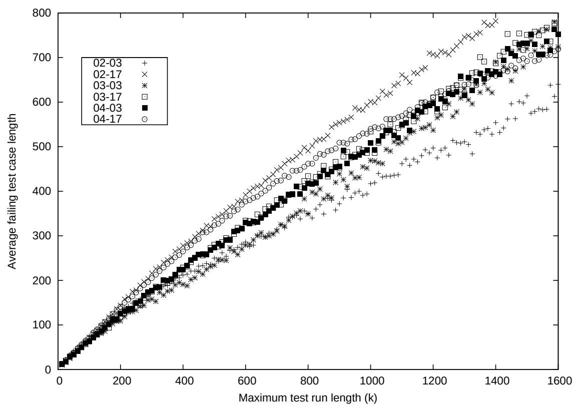

Fig. 17. Avg. failing trace length, 02-03, 02-17, 03-03, 03-17, 04-03, 04-17

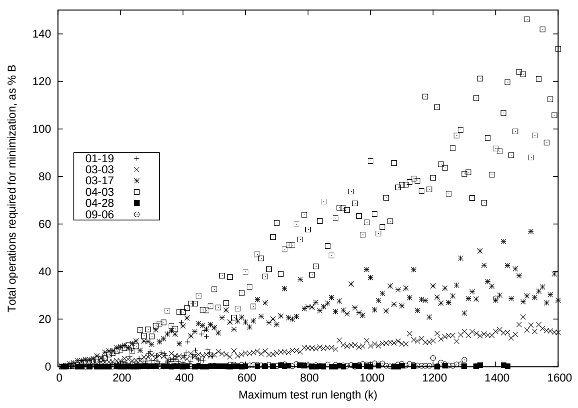

Fig. 19. Total minimization cost as % B, 01-19, 03-03, 03-17, 04-03, 04-28, 09-06

B. Failing Trace Length and Delta-Debugging

Figures 16 and 17 show how run length related to the length of failing traces. Only the results for 01-19 show any surprising features. For 01-19, we believe that one fault, with high probability of appearing just after initialization and zero probability thereafter, is responsible for the very low average up through $k = 100$. Thereafter, another failure resulting from a different fault became possible and rapidly increased average length. To some extent, this makes 01-19 less useful for predicting fault detection.

The change in trace length affected delta-debugging cost and effectiveness. As Figure 18 shows, the cost of delta-debugging does increase with failing trace length (we only report costs for a sample of versions, as delta-debugging all traces for versions with more failures proved too expensive) — note that this is a graph over trace length, not k. For very low density versions the increase with trace length was most extreme, but the number of traces to minimize so small that delta-debugging costs never amounted to more than 3% of the budget. However, as Figure 19 shows, when random testing finds more failures the cost of delta-debugging all traces can be quite significant — rising to almost 150% of the test budget for 04-03 (one and a half hours). The cost would be even higher for versions with more failures.

Figure 20 shows how the *average* length of minimized traces changed, for three medium-density versions (note the different axis for 01-19). For 03-03, the average length never exceeded 4 operations, and for the other two versions the average remained below 15 operations. For these versions of the file system, delta-debugging “flattens” increasing trace length, and k has little effect on the quality of test cases provided to developers (it may slightly *improve* with rising k, for 03-17, because of a larger pool of traces). On the other hand, Figure 21 shows the length of the *smallest* minimized test for the two lowest failure-density versions of the software — perhaps the most effective measure of the quality of traces for debugging purposes (not reported for the other versions as the shortest length is actually a constant for those versions). Here there is a much more significant relationship between the test run length and the size of minimized traces. The best failing trace ranges in size from 3 operations to 321 operations, depending on how we divide up our budget: a poor choice of k here can considerably increase the difficulty of debugging.

IV. Case Studies: Data Structures

The results in this section were based on the random testing of two data structure units: the `MoneyBag` unit from version 3.8 of the JUnit distribution [14], and a version of the TreeMap

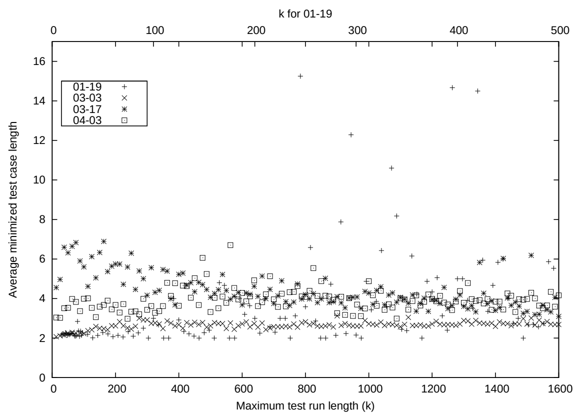

Fig. 20. Avg. minimized trace length, 01-19, 03-03, 03-17, 04-03

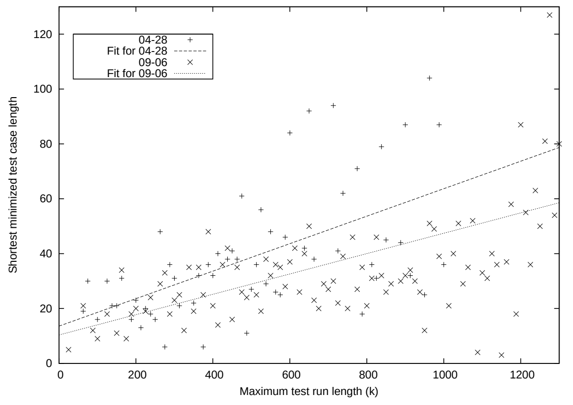

Fig. 21. Shortest minimized trace length, 04-28, 09-06 ($0.05x + 13.57$, $R^2 = 0.33$) ($0.04x + 10.34$, $R^2 = 0.15$)

unit from the Java 1.4.2 distribution into which a fault had been introduced. In each of these settings, an operation is a method call.

A. MoneyBag

The MoneyBag unit distributed with JUnit represents varying amounts of money in different currencies. The version distributed with version 3.8 of JUnit contained a fault involving the interaction of two methods. The appendTo method, a kind of specialized addition operation, created a MoneyBag with only one currency under some circumstances; however, the equals operation expected all MoneyBag objects to have more than one currency, leading it to judge two objects to be unequal when they should have been equal.

Using the framework with which the fault was originally found, we ran 10,000 test cases of each length from 5 to 420 in increments of 5. We measured the number of failures detected by the framework for each k, and also counted the number of method calls actually performed, taking into account the fact that the test case could fail before the requested length had been reached. We then calculated the average number of failures per method call actually performed.

The results are shown in Figure 22. The shape of the graph suggests that for the MoneyBag, any length over

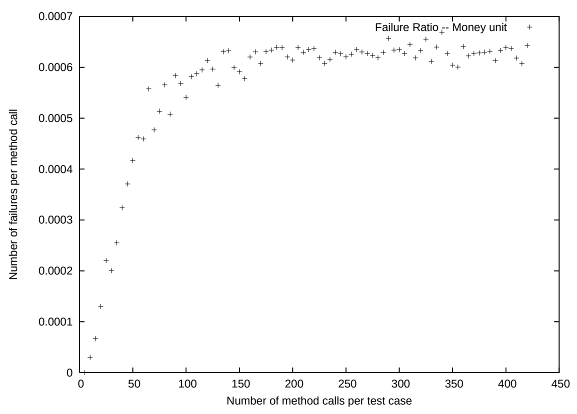

Fig. 22. Number of failures per method call, Money unit

approximately 200 results in about the same failure detection rate. It also suggests that this efficiency is optimal.

B. TreeMap

In earlier experiments [14], we tested mutants of the standard Java TreeMap unit, a red-black tree implementation, by calling random methods with parameters drawn randomly from given ranges. The driver tended to cause the size (number of keys) of the TreeMap being tested to gradually increase until it stabilized at a level where the driver was as likely to remove an element as it was to add a new element.

We wanted to simulate a fault in which a lower test sequence length was more efficient at forcing failure. We therefore seeded a fault into the code in TreeMap which caused it to fail when trying to find a key in an empty tree. We hypothesized that the failure was more likely to occur at lower test lengths, since at high test lengths the driver would have to remove all keys first. We ran 200 test cases of each length from 1 to 196 in increments of 5, and measured the number of failures per actual method call, as we had done with MoneyBag. Figure 23 shows the results. The unit was much more likely to fail when the number of operations was low, but the number of failures per method call appeared to eventually stabilize at about 0.025, one failure for approximately every 40 method calls. The graph is most similar to the file system results from 01-19 (Figure 8).

V. Analysis of Results

We formulate two hypotheses, together capable of explaining our results. The first hypothesis is that the test procedure in some cases behaves as a Markov chain; we show that this is sufficient to explain the buffer overflow and data structures results. We begin with another more general analysis than the binomial approach in Section II, based on Markov chains, of the probability that a test case will fail and the average length of a failing test case. Our second hypothesis is that the test approach in the file system cases can induce a probability of failure that can appear linear in the number of operations, which explains apparently anomalous results for some file system versions.

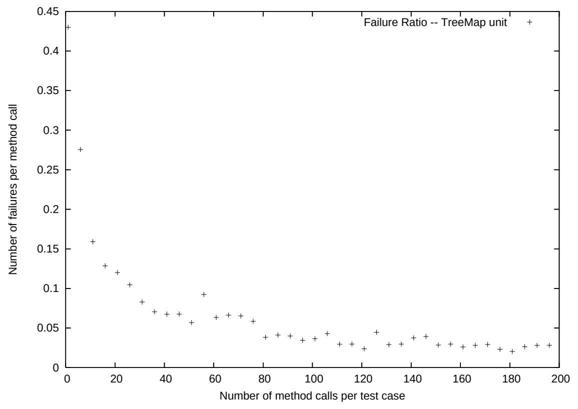

Fig. 23. Number of failures per method call, fault-seeded TreeMap unit

A. Probabilities of Failure

We assume that the random test (the test harness plus tested program) is in some state from a finite set $S = \{s_1, s_2, \dots, s_n\}$ of states. At each state $s \in S$, there is a probability $\phi(s)$ that the next operation will cause an observable failure. States $s \in S$ may not be equally likely after every operation. We use the notation $p(s, j)$ for the likelihood that the system is in state s after operation j. The probability $pf(j)$ that a failure occurs at operation j is therefore the sum, over all states s, of $p(s, j) \cdot \phi(s)$.

This can be used to calculate the average length of a test run. The system checks an oracle after each operation, and stops the sequence if it detects a failure. If no failure is detected, then the length of the sequence performed is k; if a failure is detected, then the length is m where $m \leq k$. The probability of no failure being detected is $\prod_{j=1}^{k}(1-pf(j))$; the probability of a failure being detected at operation i is $pf(i)\cdot\prod_{j=1}^{i-1}(1-pf(j))$. Therefore, when a sequence of k operations is requested, the average length $al(k)$ of the actual test case performed is $k \cdot \prod_{j=1}^{k}(1 - pf(j)) + \sum_{i=1}^{k}(i \cdot pf(i) \cdot \prod_{j=1}^{i-1}(1 - pf(j)))$. The difference in average length when the test run length is k vs. $k + 1$ operations can be denoted by $al(k + 1) - al(k)$; this expression simplifies to $\prod_{j=1}^{k}(1 - pf(j)))$.

B. Markov Chain Hypothesis

Under certain reasonable assumptions, a random test can be considered to be a Markov chain. Here we show that this hypothesis explains why the number of failures per actual operation performed stabilizes at a constant value in the buffer overflow and data structures testing.

A Markov chain [15] consists of a finite set $S = \{s_1, s_2, \dots, s_n\}$ of states, and a probability P_{ij}, for each i, j such that $1 \leq i \leq n$ and $1 \leq j \leq n$. P_{ij} represents the probability that the next operation will cause the Markov chain to make a transition from state s_i to state s_j. The Markov chain is *time-homogeneous* if the probabilities P_{ij} do not change over time. In the rest of the paper, we will assume all Markov chains are time-homogeneous.

In most circumstances, it is reasonable to make the hypothesis that a system for random testing acts as a Markov chain, in which the states represent the state of the memory and disk data that the software has access to, and the transitions represent the randomly-selected operations. Situations in which this Markov chain hypothesis does not hold include situations in which the software has an effectively infinite set of states, such as when the program is able to access and change data on an unlimited number of networked machines. These situations, however, are rare in testing environments.

Time-homogeneous Markov chains always approach *equilibrium*. That is, for each state s_i there is a *stationary state probability* π_i, such that the probability that the system is in state s_i after a transition approaches π_i; in symbols, $\lim_{j\to\infty}(p(s_i, j)) = \pi_i$.

Under the Markov chain hypothesis, the probability $pf(j)$ that a failure will be detected at operation j must converge toward a constant. Since we have seen that the difference $al(k + 1) - al(k)$ is $\prod_{j=1}^{k}(1 - pf(j)))$, we have two cases.

First, if the constant that $pf(j)$ converges toward is 0, then it is possible that $al(k + 1) - al(k)$ converges to a nonzero value as k increases. This means that test runs of requested length k increase in actual length as k increases, because a failure is less and less likely to occur as k increases — when none of the Markov chain states s_i for which $\pi_i > 0$ are states in which a failure can occur on the next operation.

Second, if the constant that $pf(j)$ converges toward is greater than 0, then $al(k+1)-al(k)$ converges toward 0. This means that test cases of requested length k typically converge in actual length to some constant as k increases. This situation happens when there is always some accessible state in which a failure can occur.

However, since we stop a test case when the first failure is detected, we note that the number of failures per operation when length k is requested is $1/al(k)$. Therefore in both of the cases above, the number of failures per operation converges to a constant: when $al(k)$ grows without bound, the constant is 0, and when $al(k)$ converges to c, the constant is $1/c$. The Markov chain hypothesis is therefore sufficient to predict that as k increases, the number $fpo(k)$ of failures per operation converges to a constant. Formally, it predicts that there is a constant c such that for all positive real numbers ϵ, there is a length k such that $|fpo(k) - c| < \epsilon$.

Although the Markov chain hypothesis predicts that $fpo(k)$ will converge to a constant, it does not predict whether this constant is a maximum of $fpo(k)$ over all k, a minimum of $fpo(k)$ over all k, or something in-between. Our empirical results show two of the behaviors. In the graphs of $fpo(k)$ for the buffer overflow example and the `MoneyBag` unit, the asymptotic value was a maximum. In the graph of $fpo(k)$ for the seeded *TreeMap* fault, the asymptotic value was a minimum. As discussed below, the asymptotic argument does not hold for the file system.

This behavior reflects the possible probabilities of failure at different distances from the start of the test case, and appears to be independent of the number of faults. For example, a graph of $fpo(k)$ that starts at 0, rises to a peak at $k = m$ and then tails off to a lower constant may reflect only one fault, if

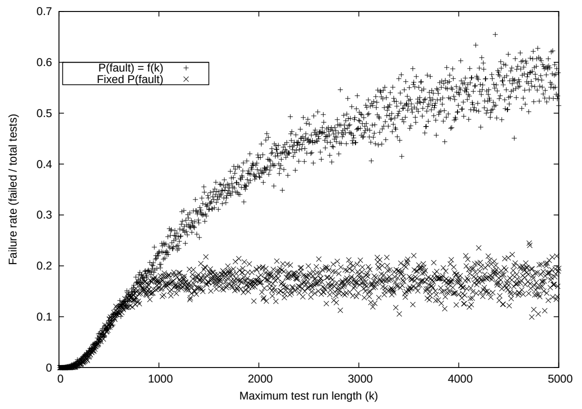

Fig. 24. Failure rates, file-system model with fault injection

the probability that we are in a state in which the fault can be triggered is highest at m; however, it may reflect two different faults, one of which can only occur early in the sequence, and the other of which can happen at any time.

C. *Linear Failure Rate Increase*

Three of the file system versions show a failure rate that seems to increase linearly with k. We attribute this somewhat unexpected behavior to a violation of **Assumption 2**: the test framework adjusts the probability of hardware faults such that the expected total number of faults per test is constant for all k, in order to avoid terminating tests early due to device failure. The underlying Markov model is therefore different for each k. If discovering an error depends on (1) simulating a certain number of faults and (2) performing operations after the faults, the influence of k on the probability of faults apparently produces a binomial distribution, but extends the range over which the graph of failure rates may appear linear, a likely explanation for Figure 15. Figure 24 shows failure rates for a toy model, featuring a single type of hardware fault. To expose the "error" a trace must first enter a state with exactly 3 faults and then perform, in order (with other operations allowed in between), 3 (out of 10) operations. The graph compares failure rates for this program if the tester (1) fixes the probability of hardware faults, $P(fault)$, or (2) adjusts $P(fault)$ with k so that 4 faults are expected in k operations.

VI. Conclusions and Future Work

The most important lesson for practitioners and random testing researchers is that run length has a major influence on the effectiveness of random testing for interactive programs. For all programs we tested, changing run length could increase the number of failures found by an order of magnitude or more. Run length also controlled the quality of failing traces produced. Delta-debugging, in many cases, reduced the importance of run length, but the cost of delta-debugging increased with length, in some cases consuming more operations than testing itself. Our study shows that the optimal run length and relationships between run length and quality and cost of minimized failing traces varied dramatically, even during the development cycle of a single program, but fit into a small number of patterns.

We also note that behaviors were generally continuous: for programs where failures are even moderately probable events, an iterative deepening approach to finding a "good-enough" run length would appear to be practical. We will investigate such an approach as future work, using both our historical data and new testing projects — in particular, we are concerned that the cost of experiments to determine failure detection for small k may result in an overall less efficient use of the budget than simply choosing a larger, sub-optimal, k in the first place. We hope to investigate how good testers are at guessing a good k — in the file system case, the difference in failure detection between the optimal point and the somewhat arbitrary choice used ($k = 1000$) was often too small to justify expensive experimentation (from 02-17 to 04-17) but was occasionally quite high (a factor of two for 02-03, and over two orders of magnitude for 01-19).

Acknowledgments. The authors would like to thank Carlos Pacheco, Gerard Holzmann, Rajeev Joshi, and Klaus Havelund for helpful comments. We also thank NSERC for support.

References

[1] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[2] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *International Conference on Software Engineering*, 2007, pp. 621–631.

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[4] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw., Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, 2004.

[5] J. Andrews, F. Li, and T. Menzies, "Nighthawk: A two-level genetic-random unit test data generator," in *Automated Software Engineering*, 2007, pp. 144–153.

[6] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28(2), pp. 183–200, 2002.

[7] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *International Symposium on Software Reliability Engineering*, 2005, pp. 267–276.

[8] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *International Symposium on Software Testing and Analysis*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 84–94.

[9] J. A. Whittaker and M. G. Thomason, "A Markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, 1994.

[10] R.-K. Doong and P. G. Frankl, "The ASTOOT approach to testing object-oriented programs," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 2, pp. 101–130, April 1994.

[11] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[12] G. Lowe, "An attack on the Needham-Schroeder public-key authentication protocol," *Information Processing Letters*, vol. 56, no. 3, pp. 131–133, 1995.

[13] http://mars.jpl.nasa.gov/msl/.

[14] J. H. Andrews, S. Haldar, Y. Lei, and C. H. F. Li, "Tool support for randomized unit testing," in *Proceedings of the First International Workshop on Randomized Testing*, Portland, Maine, July 2006, pp. 36–45.

[15] H. M. Taylor and S. Karlin, *An introduction to stochastic modeling*, 3rd ed. San Diego: Academic Press, 1998.